# Modeling and Complexity Comparison of Different Designs

Cheolgi Kim, Abdullah Al-Nayeem, Heechul Yun, Lui Sha
Department of Computer Science, University of Illinois at Urbana-Champaign

## 1 Introduction

While designing complex systems, there are many design alternatives. The choice of a simple design at the first place is crucial because it determines maintenance costs in future. The current practice is to follow good heuristics and design patterns that have stood the test of time. However, there are similar patterns to be chosen and different ways to assemble, modify and adapt the chosen patterns.

We have three related questions raised to select the simplest possible design that meets all the requirements: First, what should be the complexity metric? We suggest to used the size of the state space in verification as the metric. Some engineers claim a verification space may not be definable because of state space explosions that they commonly observe in many system verification practices. However, it is sensible to assume that the verification spaces are assumed to be measurable in our applications since the majority of avionics software is either safety critical or mission critical.

Second, how do we evaluate the (verification) complexity of alternative designs? Since verification state space size may vary by different verification methodology, we recommend the following steps to obtain a meaningful comparison we require : (a) use the same model checker as the"ruler"; (b) minimal, verifiably correct and homogeneous abstractions must be employed since abstractions reduces the model checking state space; and (c) the constructs in the chosen model checking language should closely correspond to the programming language construct of the design to limit variances caused by different modeling practices.

Third, developing accurate formal models for software designs can be time consuming. How can we reduce the overall cost and efforts? Our cost minimization approach follows classic optimization method known as branch-and-bound. This approach is discussed in detail in Section 2

To illustrate our approach, our first example is an Active-Standby control system described in Section 3 to show how to achieve consistent distributed views and actions in a networked real-time control system. First, we note that there is a generic formal low-complexity design pattern called Physically

Asynchronous Logically Synchronous architecture (PALS) and its library called PRISM that can be adapted to support the specific active-standby design. PALS/PRISM provides the advantages of a logical synchronous system on physically distributed systems. With this technique, designers can reuse their synchronous design without requiring a single physical global clock. As a result, complexity of distributed computation can be reduced to be equivalent to the synchronous design that uses a global clock. PALS and PRISM are described in more detail in Section 4.1.

Second, we compare a active-standby design using PALS/PRISM and a design based on asynchronous communications. We use model checking techniques and use the number of reachable states in each model as a metric of software complexity. To help make sure the fidelity of the model, we will show the one-to-one correspondence between the models and actual C implementations. In addition, we use two model checking tools, Maude[1] and NuSMV[2], each of which has its advantages and limitations. In the next version of this report, we will provide more examples.

The rest of this document is organized as follows: Section 2 describe our complexity metric and comparison methodology. Section 3 describes an active-standby system example. In Section 4 and 5, we describe the PALS/PRISM based synchronous design and the event-driven asynchronous design of the active-standby system. In Section 6 and Section 7, we describe our modeling technique using Maude and NuSMV. Section 8 presents the conclusion.

# 2 A Comparison of the Complexity using Model Checkers

Model checking is a formal verification technique that systematically explores the state spaces and check the design correctness. While this technique is usually used to check required properties in a design, we also use the number of reachable states in model checking as a metric for comparing complexities of design alternatives. Our analysis is viable based on the fact that safety critical systems have very strict certification requirements. For example, DO-178B certification requires full coverage testing for Level A and high coverage testing for Level B software [3]. Such coverage can be achieved only if the verification complexity is bounded.

**Modeling methodology** In order for fair comparisons, the modeling must be carefully performed. The modeling of the design consists environment modeling and computation logic modeling. The environment includes processes for dealing with task failure, packet delivery, message queues, etc. Since the environment usually have regular patterns, they can be reused over and over again once they are modeled properly. On the other hand, the computation logic is what programmers or designers create. Our approach in modeling computation logic is maintaining one-to-one correspondence between the model and its con-

crete implementation (or prototype). It means that our model closely resembles the concrete implementation both syntactically and semantically. For example, Figure 1 shows a part of C implementation and its Maude model of an event handling logic used in our case study. While language keywords used in Maude model may differ from the C code, there is clear one-to-one correspondence between the two.

The reason why we highlight the one-to-one correspondence is that this would increase confidence that any design abstraction used by the modeler would match to the real implementations. Otherwise, problems discovered in one may not exist in the other, which means that the verification effort is wasted.

```
void handle_manualselect() {
   event_t evt;
   if( my_side == STANDBY ) {
      my_side = ACTIVE;
      SENDTO_SIDE2
         ( EVT_STANDBY );
   } else {
      SENDTO_SIDE2
         ( EVT_MANUALSELECT );
   }
}
```

C code

```
eq task side1HandlesManualSelection =
   if task . mySide == STANDBY then
      task lets mySide <- ACTIVE
           sends EvtStandby
               to side2
   else
      task sends EvtManualSelection
               to side2
   fi .
```

Maude code

Figure 1: One-to-one correspondence between the C implementation(left side) and the Maude model(right side)

**Design selection procedure** Modeling a design is a iterative and time consuming procedure. Therefore, when we compare multiple designs, it may take enormous time and effort. Here, we intuitively explain our branch-and-bound design selection procedure to save such time and effort. Suppose that we have three initial design candidates $A$, $B$, and $C$. The ascending order of complexity by intuition is $\langle A, B, C \rangle$. We may start modeling $A$ first. After some modeling and design-refinement iterations, we obtain a final design $A'$ (branching). Let $|A'|$ denote the state space size of design $A'$. We then start the same process for the initial design $B$. When we model design $B$, suppose that we find $|B| \gg |A'|$. Since design refinement typically increase the state space size, sticking to design $B$ is of no use as long as we are looking for a simplest design. Therefore, we stop refining design $B$ (bounding) and move on to design $C$ to save time and efforts. We show this procedure as a flowchart in Figure 2.

Based on our experience in our case study, we believe this procedure is effective. We now describe our experience beginning from the following section.

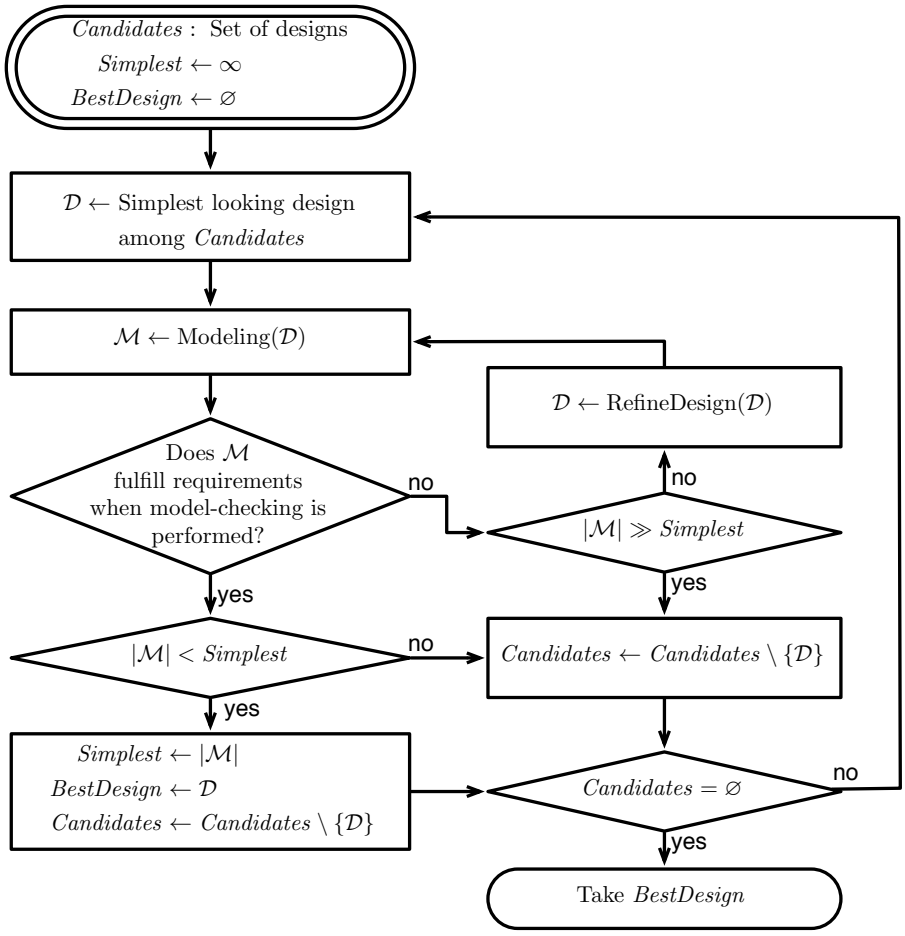



Figure 2: Branch-n-Bound design selection flowchart

# 3 Active-Standby System

In this section, we describe our case study example, called the Active-Standby System. In the active-standby system, there are two controllers: Side1 and Side2 for fault-tolerance. We assume a fault-tolerant real-time communication channel exists between these two sides. At any point in time, only one controller operates in the `ACTIVE` mode, while the other one operates in the `STANDBY` mode. In this example, the console receives commands from users and sends them to the controllers. For the sake of complexity, we only consider one command, `ManualSelect`, that toggles the active and standby modes of the two controllers. Both sides also exchange their status or heartbeat at regular intervals. Figure 3 shows the high-level structure of the active-standby system that consists of three components: Console, Side1, and Side2.

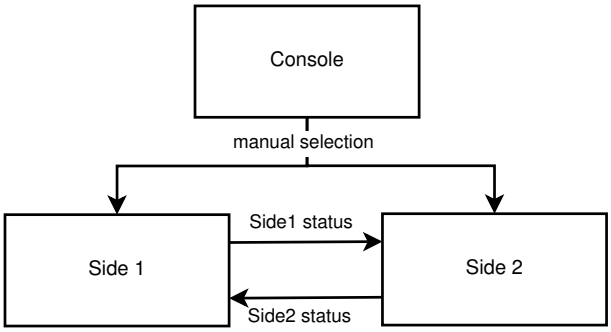



Figure 3: Active Standby System

Additionally, we assume that the controllers can fail in the fail-stop manner and also recover from such failure. In this example, both sides have to work consistently while facing random failures and user commands as long as both sides do not get failed together.

We verify our models against the following set of requirements:

1. Both sides should agree on which side is active and only one side must be active at *any point of time*.
2. If a side fails, the other side should become active.
3. The user can always choose to toggle the active side given that both sides are alive or in operative modes.

# 4 Synchronous System

One example design of the active-standby system for complexity comparison is a synchronous system design. If we have a global clock synchronizing the operations of distributed systems in lock steps, system design becomes much simpler. Designers are also less likely to make errors due to the absence of any non-deterministic asynchronous interactions.

One of the benefits of synchronous system design is that a synchronous system design can easily provide consistent views, consistent actions and synchronized state transitions across distributed nodes, which are desirable in networked real-time systems, e.g. avionics systems. For example, in a dual-redundant active-standby system, both controllers must consistently perform discrete mode controls, for example, changing the active side, in the presence of random hardware failure and asynchronous commands.

However, achieving absolute synchronization by a global clock is not physically possible in a distributed system. The clock errors of physically separated nodes can be bounded, but cannot be eliminated. While system engineers may

start the design phase with a globally synchronous model, the design must be modified to cope with the real asynchrony that exists in the system. Such modifications are usually non-trivial, and can introduce many errors. Also, one of the biggest problems is that these transformations are currently performed in an ad-hoc manner.

The rest of the section illustrates our solution to solve these problems. We present the design of the active-standby example to illustrate our points.

## 4.1 Overview of the PALS and PRISM solution

In our recent work, we developed an architectural pattern that we named *Physically Asynchronous Logically Synchronous (PALS)* system [4, 5, 6, 7] for achieving *optimal logical synchronization* in hard real-time distributed systems. Using PALS, designers can *systematically* reuse synchronous solution on physically asynchronous architecture without any changes in the application logic, even in the absence of a global clock. Also, our solution has been shown to be optimal; it is impossible to have faster rate to reach a consensus.

We have also developed a library, called PRISM, to implement such a PALS system. The PRISM library can guarantee that the views, actions and state transitions of a PALS system are exactly identical to those of a perfectly synchronous system. PALS (and PRISM) based solution can essentially reduce the design and verification costs of a complex asynchronous system so that it matches the cost of the simpler synchronous design.

The intuition of the PALS/PRISM solution is quite simple and can be illustrated in Figure 4. The basic idea is to *clock* the distributed computation logic in a logical period $T$. For example, each node in Figure 4 accepts inputs from the other nodes and the environment, computes its new local state, and generates outputs to the other nodes and environment at each step. Conceptually, this is very similar to how the asynchronous logic of an integrated circuit is made synchronous through the introduction of a clock signal that initiates the next computation step. To work correctly and preserve logical equivalence with a synchronous system, we must ensure that

1. The global computation logic for each node executes with period $T$ with a bounded error to the same global time as every other node. Note that only the distributed computations run at this period. The local control computations can run as fast as possible.

2. The period $T$ is long enough to ensure that all messages have arrived at their destination before the next computation step begins.

3. All nodes that have common external inputs start each synchronization step with identical values for those inputs.

If these constraints are met, then the behavior of this distributed system at the end of each step is equivalent to that of a synchronous system of Figure 4.

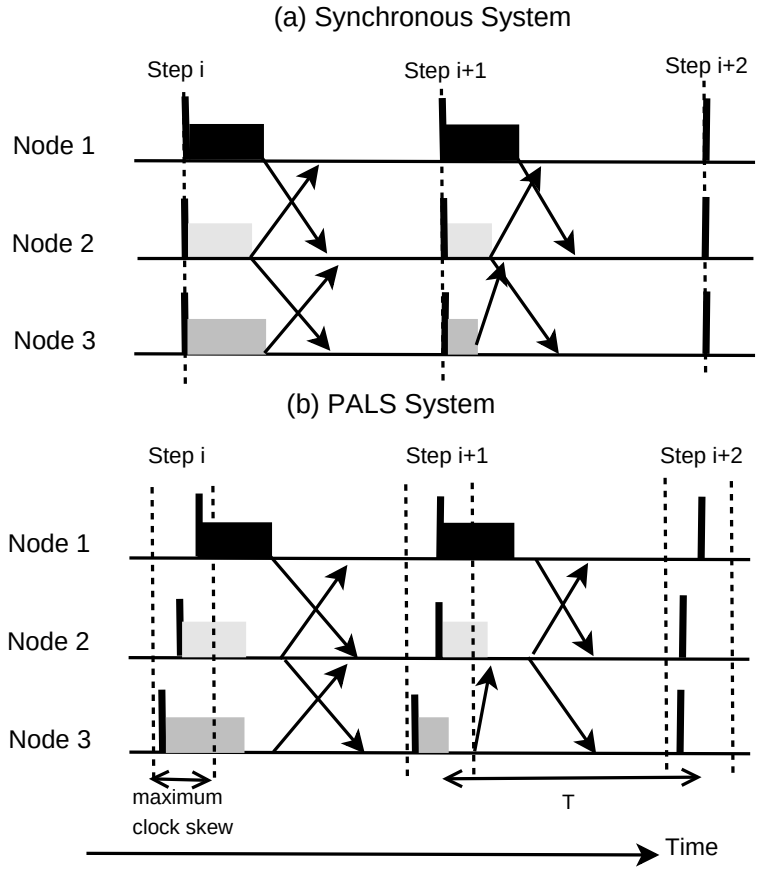



Figure 4: Synchronous system vs. asynchronous system implementing the PALS pattern

## 4.2 Synchronous Solution Based on PRISM

In the synchronous model, both Side1 and Side2 make decisions in a lock-step manner based on a common global clock. Any update to the current clock step is visible to the all components on the next clock step. Each controller manages its own mode, `ACTIVE` or `STANDBY`, and broadcasts its mode at each clock step. Since the information is distributed over the system, the mode of each controller can be treated as part of the global system state.

Figure 5 shows such synchronous operations that is also supported by PRISM. For example, Side1 posts its status `g_side1` to Side2 in clock step $i$. (`g_` is the prefix for global information.) This change is visible to Side2 on the next clock step $i+1$. For consistency, PRISM defers making the same value to be available to Side1 at step $i+1$, too. This deferral of availability simplifies the design because every node see the same picture at the same time. Manual selection

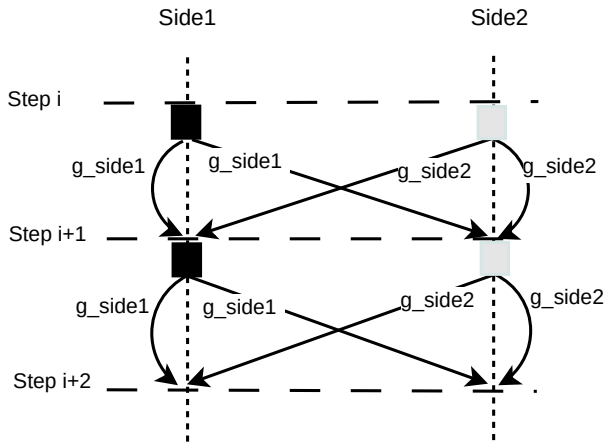



Figure 5: Synchronous execution in PRISM

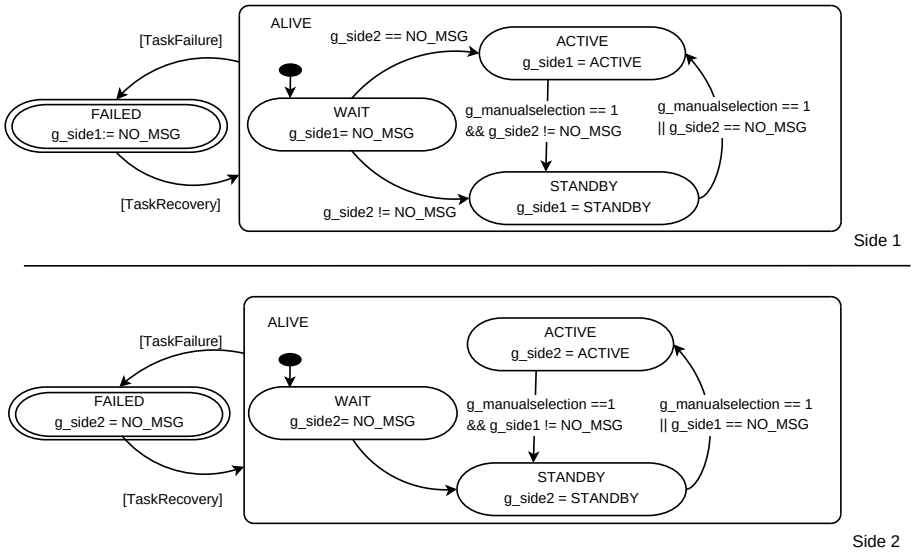
from the console is also maintained as a global information clock-by-clock, too.

Figure 6 (a) shows the state transition diagram of both Side1 and Side2 of the active-standby system design with PRISM. The logic for Side1 and Side2 are basically the same as follows:

- A controller becomes active if itself is alive and the other is not.
- If both are alive, the mode is flipped by `g_manualselect`.

There is only one difference between Side1 and Side2 logic is the (re)initialization step. If both become alive in the same step, we need a break even rule. Thereby, in such a case, Side1 in enforced to be `ACTIVE`, while Side2 is chosen to be `STANDBY`. It is defined in the state transitions from the WAIT states of both controllers.

This simple and intuitive state diagram can exactly be reflected in the global computation logic by using the PRISM library. Figure 6 (b) shows the C code for our PRISM-based implementation of the active-standby system. In this C code, each state transition happens periodically at the logical clock period as mentioned in Section 4.1. PRISM also abstracts away the asynchronous communication and presents library functions to read any global system information and post any value to it. For example, Side1 can post its status, `g_side1` of current logical clock step by using the `post_side1()` function. Then at the next logical clock step, both Side1 and Side2 can access this value consistently through the `read_side1()` function.




(a) State chart diagram

```
Side1::eachPeriod() {
  if( read_side1() == WAIT
      && read_side2() == NO_MSG ) {
    post_side1( ACTIVE );

  } else if ( read_side1() == WAIT ) {
    post_side1( STANDBY );

  } else if ( read_side2() == NO_MSG ) {
    post_side1( ACTIVE );

  } else if( read_manualSelection() ) {
    post_side1( flip( read_side1() ) );

  } else
    post_side1( read_side1() );
}
```

(a) C code for Side1

```
Side2::eachPeriod() {


  if ( read_side2() == WAIT ) {
    post_side2( STANDBY );

  } else if ( read_side1() == NO_MSG ) {
    post_side2( ACTIVE );

  } else if( read_manualSelection() ) {
    post_side2( flip( read_side2() ) );

  } else
    post_side2( read_side2() );
}
```

(b) C code for Side2

(b) C implementation

Figure 6: Design and implementation of PRISM-based active-standby system

# 5 Asynchronous Design

To compare the complexity, we also designed an active-standby system based on asynchronous communications. Since messages can be issued and delivered at any time in asynchronous communications, each node must employs message queues not to accidentally lose messages. The system configuration for active-standby system is depicted in Figure 7. In asynchronous systems, message queues are usually the source of complexity as well as the source of the state space explosions. Since messages are often queued (and hence delayed), different subsystems tend to have different perspectives on the situation. If messages are queued in both directions between a pair of nodes, the system experiences race conditions.

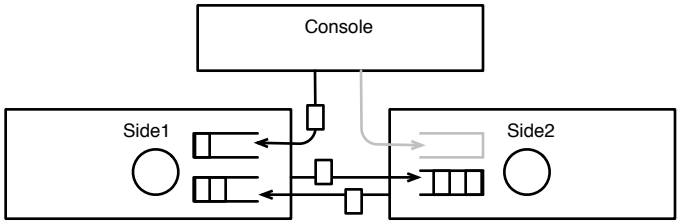
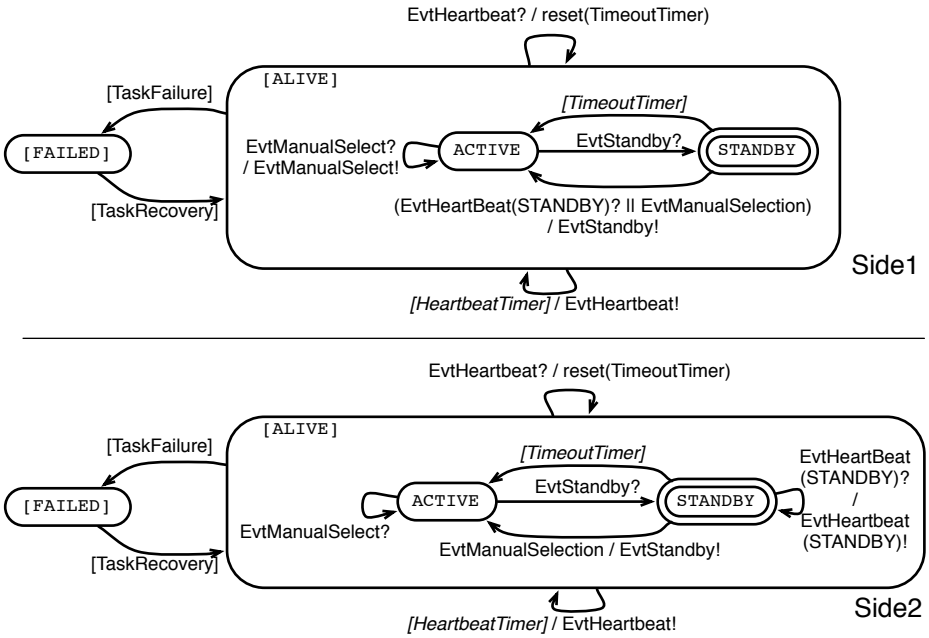



Figure 7: Active-standby system configuration in an asynchronous environment

Our asynchronous design is based on [8], which presents the design and verification of an asynchronous active-standby system. The authors claimed that they performed model-checking to verify the correctness of the system, and it took 35 hours in NuSMV [7]. Moreover, the authors mentioned that the design took about 6 months to be fully completed and verified. Since the document does not include the precise design of the system, the details of our design may not be the same as [8]. Indeed, our current design is not complete with a proof of safety. This design has the following basic behaviors:

- An alive controller generates `EvtHeartbeat` to the other controller to notify its liveness.
- If no `EvtHeartbeat` comes in for a while, the controller regards the other side has failed, taking `ACTIVE` side role.
- The console delivers `EvtManualSelect` only to Side1 at the event of user input. If Side1 is not supposed to process the event, the event is forwarded to Side2. The reason why the event is forwarded by Side1 rather than being broadcasted simultaneously to both controllers by the console is to avoid race condition.
- (`EvtManualSelect`) is only processed by the `STANDBY` controller. On the reception of the event, the `STANDBY` controller switches its role to `ACTIVE`, and enforces the other side to be `STANDBY` by sending `EvtStandby`.




(a) State chart diagram

```
void handle_timeout() {
   my_side = ACTIVE;
}

void timer_triggered() {
   SEND_HEARTBEAT_TO_SIDE2( my_side );
}

void handle_heartbeat(int state ) {
   event_t evt;
   reset_timeout();
   if(my_side == STANDBY
      && state == STANDBY) {
      my_side = ACTIVE;
      SENDTO_SIDE2( EVT_STANDBY );
   }
}

void handle_manualselect() {
   event_t evt;
   if( my_side == STANDBY ) {
      my_side = ACTIVE;
      SENDTO_SIDE2( EVT_STANDBY );
   } else {
      SENDTO_SIDE2 ( EVT_MANUALSELECT );
   }
}

void handle_recvstandby() {
   my_side = STANDBY;
}
```

C code for Side1

```
void handle_timeout() {
   my_side = ACTIVE;
}

void timer_triggered() {
   SEND_HEARTBEAT_TO_SIDE1( my_side );
}

void handle_heartbeat(int state ) {
   event_t evt;
   reset_timeout();
   if(my_side == STANDBY
      && state == STANDBY) {
      SEND_HEARTBEAT_TO_SIDE1( my_side );
   }
}

void handle_manualselect() {
   event_t evt;
   if( my_side == STANDBY ) {
      my_side = ACTIVE;
      SENDTO_SIDE2( EVT_STANDBY );
   }
}

void handle_recvstandby() {
   my_side = STANDBY;
}
```

C code for Side2

(b) C implementation

Figure 8: Design and implementation of the asynchronous active-standby system

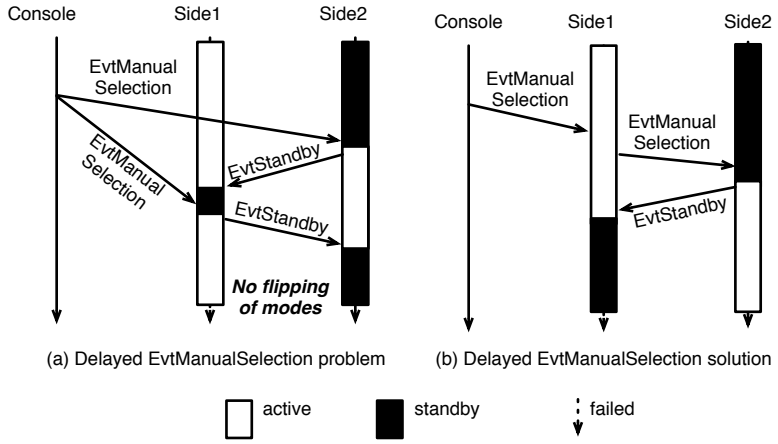



Figure 9: Delayed `EvtManualSelection` problem

- Sometimes, both side may fall into `STANDBY` mode together in the initialization. The situation is detected by the first `EvtHeartbeat`. In such a case, Side1 changes its mode to `ACTIVE` mode, and enforces Side2 to remain in `STANDBY` mode.

The design is depicted in Figure 8 (a) in a state chart diagram. The message handlers in C realizing the state chart is presented in Figure 8 (b). Even though the design looks simple, it hides complexity behind the message delays and queues.

## 5.1 Race condition examples in the asynchronous design

In fact, the design presented in Figure 8 needs to be refined and improved. However, we did not improve the design further based on branch-and-bound policy because our project goal is to identify these simplest design as soon as possible. Since the asynchronous design results in far bigger state space even with some unaddressed bugs we stopped the design refinement iterations.

In this section, we describe two race condition examples that we found during the design iterations. The first is the one resolved in the current design, and the second is not.

**Why Side1 forwards `EvtManualSelect` to Side2** As we explained earlier, `EvtManualSelect` generated by console to change `ACTIVE` controller is not simultaneously broadcasted to the controllers, but is delivered to Side1 first, and then Side1 forwards it to Side2. This design is against our intuition. In our initial design, the console was supposed to broadcast the event to both controllers. However, we found that the broadcast incurs a race condition. The current design in which Side1 forwards `EvtManualSelect` to Side2 without broadcasting is the result of refinement to avoid the race condition.

Figure 9 (a) depicts the scenario that has the race condition. Recall that the controller in `STANDBY` mode only handles `EvtManualSelect` actively in our design. In the scenario, Side2 was in `STANDBY` mode when receiving `EvtManualSelect`. Thus, it changes its mode into `ACTIVE` mode and transmits `EvtStandby` to Side1 to switch the mode of Side1 to `STANDBY`. Once this procedure is completed, Side2 becomes `ACTIVE`, and Side1 `STANDBY`, flipping their roles. The problem happens, when `EvtManualSelect` is delivered to Side1 after the above procedure is completed. The delayed delivery of `EvtManualSelect` to Side1 initiates another mode switching procedure between Side1 and Side2, rolling back their roles as if there has been no `EvtManualSelect`.

Without modeling the message delays and queues, such a race condition is not easy to detect. Hence, the state space caused by message queues and delays have to be counted as complexity sources.

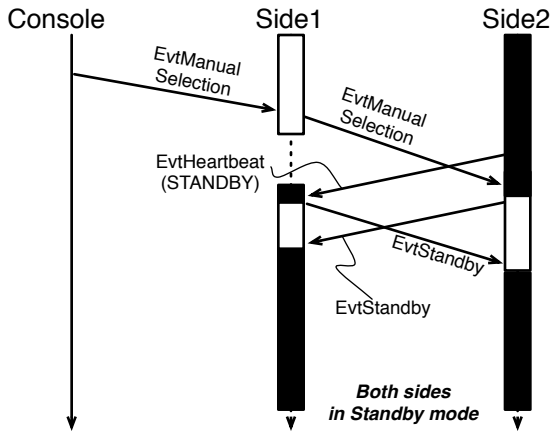



Figure 10: The next problem

**A series of coincidence making system unsafe** As we mentioned earlier, our asynchronous design described in Figure 8 is not comprehensive. There still is a scenario to make both controllers be in `STANDBY` mode in the design.[1] It is presented in Figure 10. As shown in the figure, the scenario is complicated, and series of coincidences. It is almost impossible to find such a scenario from our design without machine's assistance. Consequently, we can see that the large state space size of asynchronous system is not just a number, but the level of uncertainty that the system can fall into.

[1] Even though both controllers become `STANDBY` in the example, they can resolve the problem after exchanging heartbeat messages. However, once the model checker found this unsafe state is reachable, there can be other scenarios reaching the same state.

# 6 Model Checking using Maude

In this section, we describe how we modeled the active-standby systems in Maude, a model checking language, and how we checked the properties for the system, including state space size. Maude is a functional language that supports rewriting logic specification. While SMV and UPPAAL are designed to describe a system in a specific framework, (such as timed automata), Maude is a general purpose meta-model language. Maude is flexible and easy to extend its features because of its meta language characteristics. The maude website, `http://maude.cs.uiuc.edu`, is the best source of information about maude. See Appendix A for installation method.

| Design | Synchronous Design | Asynchronous design | |
|---|---|---|---|
| | | $n = 1$ | $n = 2$ |
| # of states | 26 | 8,122 | 212,746 |

* $n$ : the number of same kind of events allowed in a queue at an instance

Table 1: Model-checking state space size in Maude models

## 6.1 Analysis on state space results

The state space size of the synchronous and asynchronous system models in Maude is presented in Table 1. (Plase see Appendix B on how to replicate the results.) The synchronous system model has 26 states of model-checking, while the asynchronous system model has over 8,000 states even with optimistic assumptions of message queue behaviors.

The source of this state space difference is the difference of configurations. In fact, the logic of synchronous and asynchronous designs presented in Figures 6 and 8 are not very different in complexity by intuition; both have 10–20 lines of code with a couple of if statements in total. However, the complexity of the asynchronous design is hidden under the system configuration that allows messages to be queued. Recall that message queues cause race conditions, which are typically overlooked. The results means that more than 8,000 states (or 200,000) must be rigorously visited and measured to investigate if we have overlooked system states that may violate requirements in our asynchronous design.

## 6.2 Maude model walkthrough

As Maude has a distinctive grammar, understanding Maude models at the first place is nontrivial. However, we put some efforts to enhance the readability of the logic modeling part of the code, to achieve one-to-one correspondence between the model and the design. Recall Figure 1 to compare the Maude model and corresponding C code.

To briefly introduce our Maude models, let us use the Maude model for the asynchronous design coded in 'async3.maude.' You can find the following sentences at the end of the modeling file:

```
op InitialConfig : -> Configuration .
eq InitialConfig =
   < side1 : Side | alive , mySide : ACTIVE ,
                    fromOther : nil , fromConsole : nil >
   < side2 : Side | alive , mySide : STANDBY ,
                    fromOther : nil > .
```

which describes initial configuration of our model. In these sentences, we can see that the state of each task is encapsulated within `<` and `>`. The encapsulated entity is called *Object* in Maude. In the above sentences, Side1 controller is defined to be alive in `ACTIVE` mode, and has no message in the queues (`fromOther : nil` and `fromConsole : nil`), while Side2 is alive in `STANDBY` mode with no queued message.

A configuration changes its state by internal and environmental events from the defined initial configuration. State transitions are defined by the statements starting with '`rl`,' which means *rewriting logic*, for example:

```
rl [Console-ManualSelection-triggers] :
      < side1 : Side | fromConsole : nil , attr1 >
   => < side1 : Side | fromConsole : nil , attr1 >
      EvtManualSelect from console to side1 .
```

The above rewriting logic explains the state transition, at which console generates `EvtManualSelect` event. The expression before `=>` is the precondition for the event; Side1 must not have any event queued from console.[2] Since precondition does not include Side2 object, it has no precondition related to this event. As long as the precondition is satisfied, an eventual transformation from the precondition to the postcondition is allowed in the system.

Once an event occurs, it is handled by an event handler:

```
eq task side1HandlesManualSelection =
    if task . mySide == STANDBY then
        task lets mySide <- ACTIVE
             sends EvtStandby to ( task . opposite )
    else
        task sends EvtManualSelection to ( task . opposite )
    fi .
```

Even though the syntax of event handlers is very similar to that of C codes, it is not because Maude has such predefined syntax. To have such one-to-one correspondence between C and Maude codes, we have to define the required

[2]If there is no such precondition, the message queue to store the events from the console can be indefinitely long. Hence, such a message-queue restriction is required. However, such restrictions must be sensible and must not be optimistic.

syntax and utilities from the scratch. For example `lets` operation must be defined in Maude such that:

```
op _lets otherSide <-_ : Object ActiveStandby -> Object [prec 0] .
eq < s : Side | otherSide : mode1 , attr > lets otherSide <- mode2
  = < s : Side | otherSide : mode2, attr > .
```

which has no resemblance to C code in this case. Indeed, C-like code for the event handler was not given naturally. Maude language nature is far from conventional languages, so we need to put some effort to close the gap. But, we were able to achieve decent similarity with C code in the case of event handlers, because Maude is flexible and highly extensible.

# 7 Model Checking using NuSMV

NuSMV [2] is a symbolic model checker. It is capable of verifying both synchronous as well as completely asynchronous systems. In case of the synchronous system designers specify the synchronous Mealy machine representation of the system while the asynchronous system is specified as a network of non-deterministic processes. Designers can then verify the temporal properties of the system in both Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) form. The NuSMV website, `http://nusmv.irst.itc.it/index.html` is the best source of information on this model checker. See Appendix A for download and installation information.

## 7.1 Analysis of state space results

Our model checking results using NuSMV also resemble the model checking results using Maude. (Appendix B show the procedure of model checking in NuSMV). Our results show that the asynchronous model has very large state space compared to the PRISM-based synchronous models.

In our synchronous NuSMV model, there are 67 reachable states. Note that the synchronous Maude model has only 26 states. Since the input variables in the NuSMV model are considered as part of the state variables, it has has more states than the Maude model. We have also specified all requirements as CTL properties, and verified them.

In the asynchronous NuSMV model, we assumed a very restricted asynchronous communication model. The communication model consists of a globally shared message buffer of size one instead of a more realistic multiple queue model. Even with this simplification, the number of states is considerably bigger, *i.e.* more than 15 times, than the PRISM based synchronous model. Our our asynchronous model has 1017 reachable states. Any further relaxation of the assumption or the refinement of the design will less likely to improve the number of reachable state space, as shown in Section 6.

## 7.2 Brief Explanation of the NuSMV Model

The finite-state representation of each specification in NuSMV is presented as a `MODULE`. Each module defines the state variables, initial states and the state transition relations. The key part of any NuSMV specification is the transition relation. In general, designers can express any propositional expression to define the transition relation. However, the lack of powerful high-level constructs can be limiting to describe very complex interactions.

**PRISM-based synchronous model** Our synchronous NuSMV model contains three modules: `Side1Logic`, `Side2Logic` and `main` module. Every NuSMV specification must begin with a main module. There are two key state variables in this model, `g_side1` and `g_side2`, to represent the status of each side.

These state variables are updated in lock-step fashion in the module `Side1Logic` and `Side2Logic` respectively. By the properties of the PRISM based synchronous models (Figure 5), both `g_side1` and `g_side2` are observed by these two modules.

The main module in our specification has two more purposes. It randomly generates three boolean inputs for manual selection (`g_manualSelection`) and failure injection/restart command–`side1Failed` and `side2Failed`–for Side1 and Side2. The `main` module defines the constraint of no concurrent failures of both sides for `side1Failed` and `side2Failed`. Both `Side1Logic` and `Side2Logic` receive these non-deterministically generated inputs.

Based on this input, Side1Logic and Side2Logic decide the next state value of `g_side1` and `g_side2` respectively. For example, the next state transition of `g_side1` in the NuSMV is given below:

```
DEFINE
  NO_MSG := {FAILED, WAIT};

TRANS
  side1Failed = 0 & g_side1 != FAILED ->
        next(g_side1) = case
                    g_side1 = WAIT & g_side2 in NO_MSG: ACTIVE;
                    g_side1 = WAIT & !(g_side2 in NO_MSG): STANDBY;
                    g_side1 != WAIT & g_side2 in NO_MSG: ACTIVE;
                    g_side1 = ACTIVE & !(g_side2 in NO_MSG) & g_manualSelection: STANDBY;
                    g_side1 = STANDBY & !(g_side2 in NO_MSG) & g_manualSelection: ACTIVE;
                    1 : g_side1;
                    esac;
```

The propositional logic of this transition relation shows the evolution of `g_side1` in a non-failed state. It is exactly equivalent to the state diagram given by Figure 6 (a) and resembles these NuSMV `case` conditions.

In this model, we specify 6 CTL properties as shown in Section B.2 to verify the requirements. Here we summarize only three properties. The rest are similar to these three properties.

1. `AG (!ONE_ACTIVE -> ABF 0..5 ONE_ACTIVE)`: This property shows that the system can be in a state for at most 4 steps without any one side being active. It takes 5 steps to have one side as active from the initial state that includes non-deterministic restart as well as any failures during the initialization. Note that in our model both sides are initially in the `FAILED` state and start in non-deterministic order.

2. `AG ((BOTH_STABLE & FAIL_SIDE1) -> ABF 0..2 SIDE2_ACTIVE)`: If Side1 fails when both sides are stable (`ACTIVE` or `STANDBY`), then Side2 can be active in at most two steps.

3. `AG (((MANUAL_SELECTION & BOTH_STABLE) & SIDE1_ACTIVE) -> AX (BOTH_STABLE -> SIDE2_ACTIVE))`: This property states that the manual selection can switch the active mode from Side1 to Side2 only when both sides are stable.

### 7.2.1 Asynchronous model

Similar to the synchronous model there are three processes in our asynchronous NuSMV model: `Side1Logic`, `Side2Logic`, and `main`. These processes are non-deterministically executed and communicate through a single event buffer, `EVT_BUFFER`. Since it is not possible to have global consistent state information in an asynchronous model, both `Side1Logic` and `Side2Logic` maintain local copy of `g_side1` and `g_side2`.

The following messages are allowed in the model: heartbeat, standby and timeout events from Side1 and Side2, manual selection event and failure/restart events. Each of these events handles are modeled in close correspondence with the C implementation to keep the model simple. For example, consider the handler of standby event (`EVT_STANDBY_SIDE1`) in Side1. Side2 generates this event to force Side1 to become standby after the manual selection. The handler of this event in the `Side1Logic` process is given below. It updates the status variables of Side1 and removes the event from the message buffer.

```
TRANS
  ALIVE_STATE = 1 & EVT_BUFFER = EVT_STANDBY_SIDE1 ->
      next(g_side1) = ACTIVE &
      next(g_side2) = STANDBY &
      next(EVT_BUFFER) = EVT_NONE
```

We have also specified five CTL properties as shown in Section B.2 to verify the requirements. However, the properties are much weaker compared to the synchronous model. The properties are given as liveness properties, *i.e.* with eventual guarantees, because of the asynchronous interleaving of process executions.

Here we summarize only the three important CTL properties. The others are similar to these three.

1. `AG (!ONE_ACTIVE -> AF ONE_ACTIVE)`: This property states that the system can eventually reach a safe state of one active side. However, we cannot explicitly specify in how many steps will take place unlike the synchronous model.
2. `AG ((BOTH_STABLE & FAIL_SIDE1) -> AF SIDE2_ACTIVE)`: If Side1 fails when both sides are alive or stable, then Side2 eventually becomes active.
3. `AG (((MANUAL_SELECTION & BOTH_STABLE) & SIDE1_ACTIVE) -> AF (BOTH_STABLE -> SIDE2_ACTIVE))`: This property shows that the manual selection can eventually switch the active mode from Side1 to Side2 if both sides are alive or stable.

# 8 Conclusion

In this document, we demonstrated how to measure and compare the complexity of software systems. We used a simple active-standby system as a case study for these purposes. We adopted both synchronous and asynchronous design principles. For each design principle we developed a concrete C implementation and corresponding models in both Maude and NuSMV. We model-checked the desired properties and compared the state spaces. We quantitatively showed that the complexity of the asynchronous design was much higher than that of the synchronous one and also how easy it is to introduce errors in the asynchronous design, even for a very small system.

# A Installation of Maude and NuSMV

In this section, we describe installation methods of model checkers we used in this document.

## A.1 Maude

Installation of Maude is mostly about unzipping a file and setting a right path environment. Once the `maude.tar.gz` is downloaded from the website, you may want to unzip it in a directory. Suppose that we want to unzip it at `/usr/local/`.

```
/usr/local$ sudo tar xvf <Downloaded-directory>/maude.tar.gz
/usr/local$ cd maude
/usr/local$ sudo ln -s  maude maude.linux    (in linux 32bit system)
/usr/local$ sudo chmod 755 maude
```

Do not make symbolic link to run maude outside of maude installation directory. The execution file must reside in the same directory as all the other maude installation files. Then, the path environment variable must be set to include `/usr/local/maude`. For example, you can append the following line at the end of `~/.bashrc`.

```
export PATH=$PATH:/usr/local/maude
```

Now, we are ready to run Maude.

```
$ source ~/.bashrc
$ maude
     \||||||||||||||||||/
   --- Welcome to Maude ---
     /||||||||||||||||||\
    Maude 2.5 built: May  7 2010 18:55:22
    Copyright 1997-2010 SRI International
   Sun Aug  1 13:41:32 2010
Maude> _
```

## A.2 NuSMV

NuSMV is freely available. It can be downloaded from `http://nusmv.irst.itc.it/index.html`. NuSMV is available for Windows and Linux (Redhat) environment. In addition to its source, users can directly use the binaries without any further installation. The latest version is 2.5.0.

NuSMV website hosts relevant tutorials and user manual. You can download them from the following websites:

1. Tutorial: `http://nusmv.fbk.eu/NuSMV/tutorial/index.html`
2. User manual: `http://nusmv.fbk.eu/NuSMV/userman/index-v2.html`

### A.2.1 Installation in Windows

The website also gives a self-extracting installation file and allows to install the program in the system. The executable is located in `INSTALL_DIR\NUSMV\2.4.3\bin\NuSMV.exe`. It has to be run from the command prompt.

Alternately, users can download the binaries for Windows environment, unzip it and run the executable from the command prompt.

### A.2.2 Installation in Linux

**Method 1** Users can download the binary (NuSMV-2.5.0-i686-redhat-linux-gnu.tar.gz)and unzip the files [3]. The executable is located in `INSTALL_DIR\bin\NuSMV`.

**Method 2** Users can download the source files (e.g. NuSMV-2.4.3.tar.gz) from the website[4]. NuSMV source files come with additional source code of CUDD necessary for the execution of NuSMV. Additionally, it uses ZChaff which has to be download from `http://www.princeton.edu/~chaff/zchaff.`

[3] Users might need to install some shared libraries, before using the executable.
[4] I found some problems installing the version 2.5.0

html. Before installing NuSMV, users need to compile and link both ZChaff and CUDD with NuSMV. To install the NuSMV, please follow the following set of commands.

```
$ tar zxvf NuSMV-2.4.3.tar.gz
$ cd NuSMV-2.4.3
$ cd cudd-2.4.1.0
$ make
$ cd  ../zchaff
$ wget http://www.princeton.edu/~chaff/zchaff/zchaff.64bit.2007.3.12.zip
$ ./build.sh
$ cd ../nusmv
$  ./configure --enable-zchaff
$ make
$ make install
```

NuSMV is by default installed in /usr/local/bin.

# B Model Checking using Maude and NuSMV

In this section, we describe how to replicate our state space comparison result in both Maude and NuSMV.

## B.1 Maude

If you would like to reproduce the result:

1. run the maude shell:

```
$ maude
     \||||||||||||||||||/
   --- Welcome to Maude ---
     /||||||||||||||||||\
    Maude 2.5 built: May  7 2010 18:55:22
    Copyright 1997-2010 SRI International
   Sun Aug  1 13:41:32 2010
Maude> _
```

2. load a Maude model file:
   `Maude> load prism.maude` (For synchronous design)
   `Maude> load async3.maude` (For asynchronous design)

3. and search the state space:

```
Maude> search InitialConfig =>! conf .
search in ACTIVE-STANDBY : InitialConfig =>! conf .

No solution.
states: 26  rewrites: 4416 in 8ms cpu (14ms real) (526278 rewrites/second)
```

The searching statement is to search deadlock case in the model starting from the initial configuration (`InitialConfig`).

Notice that the asynchronous design has a parameter to specify a environment characteristics. To prohibit message queues from growing indefinitely, which causes unbounded model-checking space, we limits the number of events of same kind in a queue. It can be specified by changing the line of the Maude model file of

```
eq maxSameEvtAllowed = 2 .
```

into:

```
eq maxSameEvtAllowed = n .
```

where `n` is the number that you want to set.

## B.2 NuSMV

In this section, we show how the model checking results can be replicated for the PALS/PRISM based synchronous system and the asynchronous system.

**Synchronous active-standby system** The procedure of model checking of PRISM-based NuSMV model (`prism.smv`)is shown in the following:

1. run the model in the interactive mode of NuSMV.
   `c:\NuSMV-2.5.0\bin\NuSMV.exe -int prism.smv`
2. load and build the module with command `go`. `NuSMV > go`
3. check the CTL properties with command `check_ctlspec`.

   ```
   NuSMV > check_ctlspec
   -- specification AG (!ONE_ACTIVE -> ABF 0..5 ONE_ACTIVE)  is true
   -- specification AG !BOTH_ACTIVE  is true
   -- specification AG ((BOTH_STABLE & FAIL_SIDE1) -> ABF 0..2 SIDE2_ACTIVE) is true
   -- specification AG ((BOTH_STABLE & FAIL_SIDE2) -> ABF 0..2 SIDE1_ACTIVE) is true
   -- specification AG (((MANUAL_SELECTION & BOTH_STABLE) & SIDE1_ACTIVE) ->
   AX (BOTH_STABLE -> SIDE2_ACTIVE))  is true
   -- specification AG (((MANUAL_SELECTION & BOTH_STABLE) & SIDE2_ACTIVE) ->
   AX (BOTH_STABLE -> SIDE1_ACTIVE))  is true
   ```

4. print the number of reachable states with `print_reachable_states` command.

```
NuSMV > print_reachable_states
###############################################################################
system diameter: 5
reachable states: 67 (2^6.06609) out of 128 (2^7)
###############################################################################
```

**Asynchronous active-standby system** Similar to the previous example we can calculate the number of reachable states and verify the properties of the asynchronous NuSMV model (`asynch.smv`) in the interactive mode:

1. run NuSMV and load the model.

```
c:\NuSMV-2.5.0\bin\NuSMV.exe -int asynchronous-activestandby.smv
NuSMV > go
```

2. check the properties.

```
NuSMV > check_ctlspec
-- specification AG (!ONE_ACTIVE -> AF ONE_ACTIVE)  is true
-- specification AG ((BOTH_STABLE & FAIL_SIDE1) -> AF SIDE2_ACTIVE) is true
-- specification AG ((BOTH_STABLE & FAIL_SIDE2) -> AF SIDE1_ACTIVE) is true
-- specification AG (((MANUAL_SELECTION & BOTH_STABLE) & SIDE1_ACTIVE) ->
AF (BOTH_ALIVE -> SIDE2_ACTIVE))  is true
-- specification AG (((MANUAL_SELECTION & BOTH_STABLE) & SIDE2_ACTIVE) ->
AF (BOTH_ALIVE -> SIDE1_ACTIVE))  is true
```

3. print the number of reachable states.

```
NuSMV > print_reachable_states
###############################################################################
system diameter: 11
reachable states: 1017 (2^9.9901) out of 3159 (2^11.6253)
###############################################################################
```

# References


[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude – A High-Performance Logical Framework*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[2] NuSMV, "http://nusmv.irst.itc.it/index.html."

[3] RTCA, "DO-178B - Software Considerations in Airborne Systems and Equipment Certification," *RTCA Inc.*, 1992.

[4] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Ölveczky, "PALS: Physically Asynchronous Logically Synchronous Systems," University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/11897, Tech. Rep., 2009.

[5] J. Meseguer and P. C. Ölveczky, "Formalization and Correctness of the PALS Pattern for Asynchronous Real-Time Systems," University of Illinois at Urbana-Champaign, https://www.ideals.illinois.edu/handle/2142/14214, Tech. Rep., 2009.

[6] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, and D. D. Cofer, "A Formal Architecture Pattern for Real-Time Distributed Systems," in *Proceedings of the 30th Real-Time Systems Symposium*, 2009.

[7] S. P. Miller, D. D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem, "Implementing Logical Synchrony in Integrated Modular Avionics," in *Proceedings of the 28th Digital Avionics Conference*, 2009.

[8] S. P. Miller, M. W. Whalen, D. O'Brien, M. P. Heimdahl, and A. Joshi, "A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures." NASA Contractor Report CR-2005-213912, 2005.